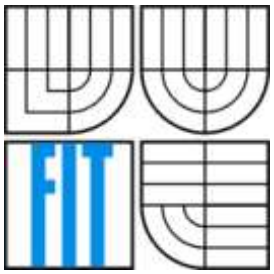
VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

Fakulta informačních technologií
Faculty of Information Technology

BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

Brno, 2016 Pavel Hamerník

VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY



FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV POČÍTAČOVÝCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF COMPUTER SYSTEMS

PROCEDURÁLNÍ GENEROVÁNÍ MĚST

PROCEDURAL GENERATION OF CITIES

BAKALÁŘSKÁ PRÁCE
BACHELOR‘S THESIS

AUTOR PRÁCE
AUTHOR

PAVEL HAMERNÍK

VEDOUCÍ PRÁCE
SUPERVISOR

ING. LUKÁŠ POLOK

BRNO 2016

Abstrakt

Tato práce se zabývá procedurálním generováním měst. Jsou v ní definovány jednotlivé etapy generování města a existující metody sloužící k jejich vytvoření. Dále jsou uvedeny postupy vycházející z použitých metod při implementaci systému schopného generovat města. Výsledný generátor je schopen vytvářet města na základě počátečního náhodného semínka.

Abstract

This work is about procedural generation of cities. There are defined the different stages of city generation and existing methods used to achieve them. The following describes the procedures used to implement a system capable of generating cities. The resulting generator is able to generate cities based on input random seed.

Klíčová slova

procedurální generování, generování města, L-systém, Perlinův šum, squarified treemap, generování interiéru, generování terénu.

Keywords

procedural generation, generation of cities, L-system, Perlin noise, squarified treemap, generation of interier, generation of terrain.

Citace

Hamerník, Pavel: Procedurální generování měst, bakalářská práce, Brno, FIT VUT v Brně, 2016

Procedurální generování měst

Prohlášení

Prohlašuji, že jsem tuto bakalářskou práci vypracoval samostatně pod vedením pana Ing. Lukáše Poloka.
Uvedl jsem všechny literární prameny a publikace, ze kterých jsem čerpal.

……
Pavel Hamerník
28.7.2016

Poděkování

Rád bych poděkoval vedoucímu mé bakalářské práce Ing. Lukáši Polokovi za odborné rady a cenné připomínky, které vedly ke vzniku této práce. Dále bych chtěl poděkovat svým rodičům za podporu nejen při vypracování této práce, ale i po celou dobu studia.

Obsah

1 Úvod

Ve většině 3D grafických aplikace, her a animací byli dříve všechny modely vytvářeny ručně. Ruční modelování při malém počtu objektů je ještě zvladatelné. Problém nastává při modelování velkého města, kde se vyskytuje velké množství budov s různými úrovněmi detailu. Manuální tvorba takového počtu objektů by zabralo velké množství času. Vývojáři a animátoři se z toho důvodu přiklání k městům, která jsou generována. Typickým příkladem hry s vygenerovaným městem včetně interiérů je hra GTA IV firmy Rockstar Games[1]. Díky procedurálnímu generování potom grafici přistoupí k hotovému modelu města a jen jej podle návrhu či fantazie poupraví. Práce se zabývá tématikou vytvoření města pomocí procedurálních metod.

Práce začíná přehledem teorie, jejíž poznatky jsou později použity při implementaci aplikace. Následuje kapitola 3, která popisuje práce zabývající se podobnou tématikou. Dále v kapitole 4 jsou popsány důležité knihovny a aplikační rozhraní, které jsou v rámci této práce využity. Stěžejní část práce se nachází v kapitole 5, kde je prezentován návrh a implementace aplikace. Kapitola 6 obsahuje závěr, shrnutí celé práce a navrhují možná rozšíření.

[1] http://www.rockstargames.com/

2 Teorie

V této kapitole je uveden teoretický základ, ze kterého práce čerpá. Jsou zde popsány postupy, algoritmy a vzorce, které výsledná aplikace používá.

2.1 Procedurální generování

Pojem procedurální generování se vztahuje ke generování obsahu za běhu aplikace. Často se tento pojem používá v počítačových aplikacích, herním a filmovém průmyslu. V těchto konkrétních oborech dokáže výrazně snížit náklady na výrobu.

2.2 Generování náhodných čísel

Generátory náhodných čísel se používají v mnoha odvětvích informatiky. Jedná se buď o funkci, nebo hardwarové zařízení. Budeme-li pracovat pouze se softwarovými generátory, tak není možné dosáhnout skutečné náhodnosti. Deterministické algoritmy, které se ke generování čísel používají, vytváření pouze pseudonáhodné posloupnosti. Tyto posloupnosti tedy nejsou skutečně náhodné, ale pouze tak vypadají. Pro zvětšení iluze se často jako počáteční stav generátoru (náhodné semínko, angl. random seed) využívá čas.

2.2.1 Kongruentní generátor

Nejznámější a velmi často používaným pseudonáhodným generátorem čísel, je lineární kongruentní generátor popsaný vzorcem

$$x_{i+1} = (ax_i + c)\ mod\ m, \tag{2.1}$$

kde operace mod znamená zbytek po celočíselném dělení. Počáteční nastavení generátoru je určeno pomocí hodnoty x_0 a nazývá se náhodné semínko. Generátor generuje pseudonáhodná celá čísla v intervalu $\langle 0, m)$. Jelikož je počet možných hodnot v tomto rozsahu omezen, začne se nejpozději po m vygenerovaných číslech opakovat stejná posloupnost (perioda generátoru). Konstanty a, c a m je nutné zvolit vhodně. Maximální možné periody je možné dosáhnout pomocí těchto podmínek

- m a c jsou nesoudělná čísla
- $a - 1$ je dělitelné všemi prvočíselnými faktory čísla m
- $a - 1$ je násobek 4, jestliže m je násobek 4

Takovéto generátory jsou jednoduché na implementaci. Mezi jejich hlavní výhody patří rychlost a paměťová nenáročnost.

2.2.2 Spojitý pseudonáhodný generátor

Největší problém využití prostého generátoru čísel je nerealistický vzhled. Obrazy vypadají uměle, protože mezi hodnotami, které jsou získány z generátoru, není žádná spojitost. S řešením tohoto problému se zabýval Ken Perlin, který vyvinul metodu zvanou Perlinův šum. Základem jeho metody je funkce, která přijímá reálné číslo, a vrací pseudonáhodnou hodnotu. Tato funkce, ale bere v potaz rozdíly na jejím vstupu. V případě, že funkce obdrží dvě velice podobné hodnoty na vstupu, pak jsou na výstupu náhodně vypadající, ale opět dvě velmi podobná čísla. Existuje mnoho různých implementací této funkce, v rámci této práce byl použit generátor definovaný rovnicemi [1]:

$$N(x) = G\big((x \ll 13) xor\ x\big) \tag{2.2}$$

$$G(x) = 1 - \frac{(60493x^3 + 19990303x + 1376312589)\ and\ 7fffffff}{1073741824} \tag{2.3}$$

kde operace xor je bitová exkluzivní disjunkce, operace and je bitový logicky součin. Hexadecimální maska 7fffffff odpovídá maximálnímu číslu, které lze uložit do 32bitového celého čísla, a slouží proti možnému přetečení hodnoty. Takto definovaný generátor přijímá na svém vstupu celá čísla a jeho výstupem jsou hodnoty v rozsahu $\langle -1; 1\rangle$. Jedná se o deterministický generátor, který pro každý vstup vrátí vždy stejný výsledek. Takto nadefinované rovnice odpovídají jednodimenzionálnímu generátoru. Pokud bychom chtěli přidat více dimenzí, můžeme toho dosáhnout například tím, že jednu z hodnot vynásobíme prvočíslem.

$$N(x, y) = N(x + 57y) \tag{2.4}$$

2.2.3 Interpolace

Interpolace je matematický proces, kdy z dané diskrétní množiny, za pomocí aproximace, vypočítáme zbylé hodnoty hodnoty funkce. Existuje několik typů interpolace, každá se liší svou přesností a složitostí výpočtu.[2]

Lineární interpolace

Nejjednodušší a nejméně náročný způsob interpolace. Jestliže jsou známy dva body $a_1 = (x_1; y_1)$ a $a_2 = (x_2; y_2)$, linerání interpolace je potom přímky mezi těmito body. Pro hodnotu x je interval $\langle x_1; x_2\rangle$. Hodnota y podél přímky je dána rovnici

$$\frac{y - y_2}{y_2 - y_1} = \frac{x - x_1}{x_2 - x_1} \tag{2.5}$$

Vyřešením této rovnice pro y, dostaneme rovnici

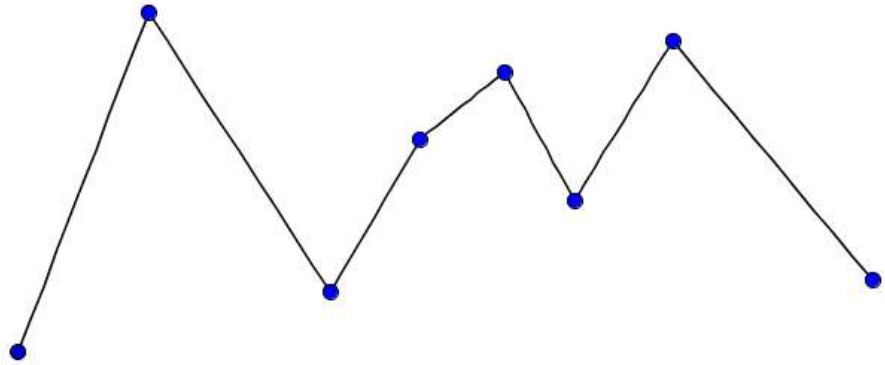
$$y = y_1 + (x - x_1)\frac{y_2 - y_1}{x_2 - x_1}, \tag{2.6}$$

která je rovnicí lineární interpolace. Tuto rovnice je možné dále zjednodušit.

$$y = (1 - t)y_1 + ty_2 \tag{2.7}$$

$$t = \frac{x - x_1}{x_2 - x_1} \tag{2.8}$$

Výhodou lineární interpolace je její málá výpočetní náročnost. Díky tomu je v praxi velmi používaná. Její velkou nevýhodou je nepřesný výpočet, přechody mezi body jsou ostré, a v důsledku není příliš spojitá.



Obrázek 2.1: Ukázka lineární interpolace na množiny bodů.

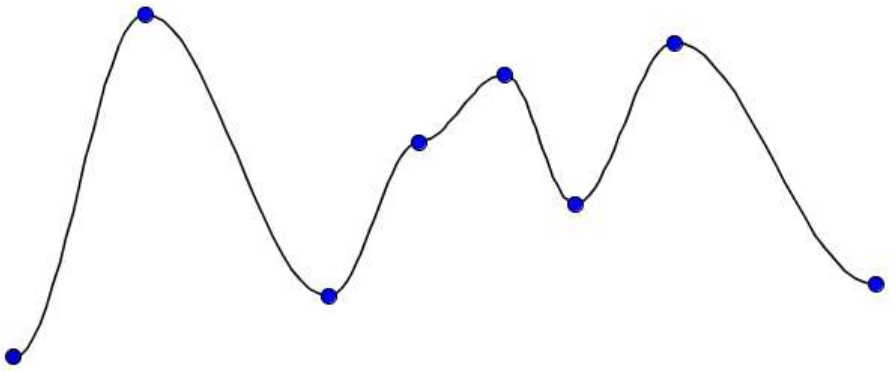
Kosinova interpolace

Vylepšením lineární interpolací, je kosinová interpolace. Tato metoda částečně řeší problém ostrých přechodů mezi body. Řešení je založeno na použití části funkce kosinus mezi dvěma body. Úpravou rovnic lineární interpolace (2.7), dostaneme interpolaci kosinovu. Tato metoda je výpočetně mírně náročnější, ale dosahuje znatelně lepších výsledků.

$$y = \big(1 - f(t)\big)y_1 + f(t)y_2 \tag{2.9}$$

$$f(x) = \frac{(1 - \cos \pi x)}{2} \tag{2.10}$$

Parametr t zůstává stejný jako u lineární interpolace (2.8).



Obrázek 2.2: Ukázka kosinovy interpolace na množině bodů.

2.2.4 Perlinův šum

Perlinův šum je pojmenovaném po svém autorovi Kenu Perlinovi [1]. Jedná se o metodu vhodnou pro generování grafického šumu. Perlinův šum nám generuje pseudonáhodné hodnoty v rozsahu $\langle -1; 1 \rangle$. Využívá spojitého pseudonáhodného generátoru popsaného výše. I přesto že tento generátor bere v potaz rozdíly v hodnotách na vstupu, tak by pouhé vyhlazení nemusel být dostačující. Například při použití dvojrozměrného šumu by na obraze při použití interpolace mohly vzniknout patrné ostré přechody. Tento problém je řešen pomocí kroku vyhlazení pomocí okolních hodnot. Namísto toho aby hodnota, která se dále bude interpolovat, byla získána přímo ze spojitého generátoru, bude získán průměr okolí požadovaného bodu. Vyhlazovací funkce pro jednodimenzionální generátor je možná popsat takto:

$$SmoothN(x) = \frac{N(x)}{2} + \frac{N(x-1)}{4} + \frac{N(x+1)}{4} \tag{2.11}$$

Podobně je možné definovat funkci i ve více dimenzích. Pro vyhlazení dvou dimenzí použijeme pro zjednodušení následující funkce C(x,y) a S(x,y). Funkce C(x,y) značí součet vyhlazovaných bodů v diagonálním směru, S(x,y) značí součet vyhlazovaných bodů ve vertikálním nebo horizontálním směru

$$C(x,y) = N(x-1,y-1) + N(x+1,y-1) + N(x-1,y+1) + N(x+1,y+1) \tag{2.12}$$

$$S(x,y) = N(x-1,y) + N(x+1,y) + N(x,y-1) + N(x,y+1) \tag{2.13}$$

$$SmoothN2D(x,y) = \frac{C(x,y)}{16} + \frac{S(x,y)}{8} + \frac{N(x,y)}{4} \tag{2.14}$$

Všechny u všech výše popsaných rovnic však parametry x a y jsou celá čísla. Abychom získali hodnotu pro reálné vstupy je potřeba rozdělit vzít nejbližší okolí hledaného bodu a hodnoty mezi sebou interpolovat. Toho je dosaženo v dvourozměrném prostoru pomocí následujícho algoritmu

```
InterpolatedNoise(x,y):
        x0 = floor(x)
        y0 = floor(y)

        diffX = x-x0
        diffY = y-y0

        s00 = SmoothN2D(x0,y0)
        s01 = SmoothN2D(x0,y0+1)
        s10 = SmoothN2D(x0+1,y0)
        s11 = SmoothN2D(x0+1,y0+1)

        s0 = interpolate(s00,s10,diffX)
        s1 = interpolate(s01,s11,diffX)

        InterpolatedNoise = interpolate(s0,s1,diffY)
end
```

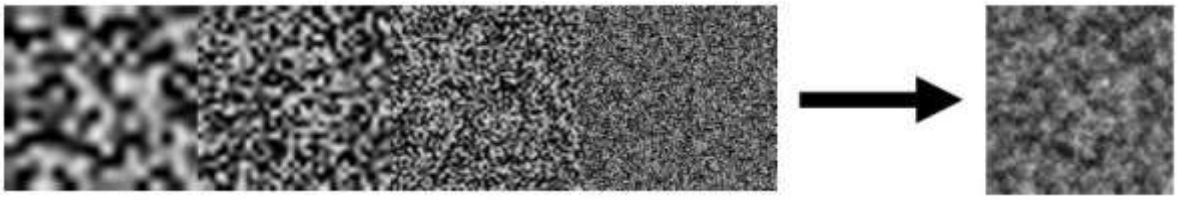
Algoritmus 2.1: Interpolace šumové funkce

Funkce interpolate ve výše popsaném algoritmu může být jakákoli interpolace například lineární nebo kosinova. Základní představa Perlinova šumu spočívá v tom, že se vytvoří několik funkcí, každá s jinou amplitudou a frekvencí a každá z nich představuje úroveň detailu výsledného obrazu. Výsledná hodnota je pak spočítána jako jejich součet. Takto složená funkce se nazývá Perlinův šum. Výsledek šumu je závislý na zvolených parametrech: frekvence, amplitudy, perzistence a počtu oktáv.

```
Nastav frekvence = počáteční frekvence
Nastav amplituda = počáteční amplituda
Nastav suma = 0
Opakuj  n-krát podle počtu oktáv
        suma += InterpolatedNoise(x*frekvence,y*frekvence) * amplituda
        amplituda *= perzistence
        frekvence /= perzistence
```

Algoritmus 2.2: Algoritmus Perlinova šumu

Výsledkem tohoto algoritmu je proměnná suma, která značí výsledek Perlinova šumu s výše uvedenými parametry. Na obrázku Obrázek 2.3 lze pozorovat Perlinův šum složený ze čtyř dílčích funkcí při různých frekvencích.



Obrázek 2.3: Ukázka složení Perlinova šumu.

2.3 Geometrie objektů

Zde jsou popsány základní geometrické objekty, jako jsou přímky, úsečky, polygony a důležité operace nad nimi. Vychází se z analytické geometrie lineárních objektů, kde se pro výpočty používají vektory v rovině nebo v prostoru. Tyto teoretické poznatky jsou v práci prakticky použity k výpočtům, jako jsou průniky silnic, normály stěn a dalších.

Vektory leží v definovaném kartézském souřadnicovém systému, který má svůj počátek v bodě, jež má všechny své složky nulové. Souřadný systém je dále definován osami, které na sebe bývají kolmé. Takový to souřadný systém nazýváme eukleidovský. Bod udává ortogonální vzdálenost od každé z os. Obecně lze vektor zapsat jako n-tici $v = (v_1, v_2, \dots, v_n)$, kde n udává rozměr prostoru, ve kterém je vektor orientován. Vektor má svou velikost a směr. Vektoru se spočítá jako rozdíl dvou bodů. Máme-li body $X = [x_1, x_2, \dots, x_n]$ a $Y = [y_1, y_2, \dots, y_n]$, pak vektor $\overrightarrow{XY} = Y - X = (y_1 - x_1, y_2 - x_2, \dots, y_n - x_n)$. Velikost vektoru v se počítá jako vzdálenost dvou bodů a značí se jako $|v|$.

$$|v| = \sqrt{\sum_{i=1}^{n} v_i^2} \tag{2.15}$$

Velikost vektoru v lze měnit pomocí násobení skalárem, což znamená násobení každé složky vektoru týmž skalárem.

$$av = (av_1, av_2, \dots, av_n) \tag{2.16}$$

Základní operace součtu a rozdílu se počítají jako operace nad jednotlivými složkami vektoru. Rovnice součtu, resp. rozdílu jsou dány následující rovnicí

$$u + v = (u_1 + v_1, u_2 + v_2, \dots, u_n + v_n) \tag{2.17}$$

$$u - v = (u_1 - v_1, u_2 - v_2, \dots, u_n - v_n) \tag{2.18}$$

Další operací je skalární součin, který udáva vztah mezi velikostí vektorů a úhlem jimi svíraným. Značí se pomocí tečky "·". Výsledkem této operace je skalár. Skalární součin mezi vektory u a v, jenž mezi sebou svírají úhel φ, se spočítá rovnicí:

$$u \cdot v = |u||v| \cos\varphi = \sum_{i=1}^{n} u_i v_i \tag{2.19}$$

Jednotkový vektor, nebo také nazýván normalizovaný, se získá dělením každé složky velikostí vektoru. Díky tomu vznikne vektor, který má směr a jeho velikost je rovna 1.

$$\hat{u} = \frac{u}{|u|} \tag{2.20}$$

2.3.1 Rovinné objekty

Rovina je definována dvěma osami x a y a lze v ní sestrojit základní geometrické objekty: bod a přímku. Pro sestrojení přímky je potřeba znát její směrový vektor.

Obecná rovnice přímky vychází z normálového vektoru, to je vektor kolmý na vektor směrový. Máme li vektor $u = (u_x; u_y)$, pak normálový vektor vytvoříme záměnou složek směrového vektoru a negací jedné z nich: $n = (u_x; -u_y) = (n_x; n_y)$. Obecná rovnice přímky v rovině má tvar

$$ax + by + c = 0 \tag{2.21}$$

Koeficienty a a b jsou určeny pomocí normálového vektor n, který je kolmý k přímce. Parametr c pak souvisí se vzdáleností přímky od počátku souřadného systému.

Obecné rovnice přímek se dají využít pro zjištění, jestli se bod v rovině leží na přímce nebo pro získání průsečíku dvou přímek. Zda bod leží na přímce, zjistíme dosazením do obecné rovnice přímky. Pokud je výsledkem rovnost pak bod leží na přímce jinak leží mimo.

Průsečík dvou přímek se řeší jako soustava dvou lineárních rovnic, kde výsledkem je právě bod který leží na průsečíku obou přímek. Máme li přímky $p: ax + by + c = 0$ a přímku $q: dx + ey + f = 0$. Pak průsečíkem těchto přímek je bod P řešen pomocí matice:

$$\begin{pmatrix} a & b \\ d & e \end{pmatrix} \begin{pmatrix} P_x \\ P_y \end{pmatrix} = \begin{pmatrix} -c \\ -f \end{pmatrix} \tag{2.22}$$

2.3.2 Prostorové objekty

V prostoru díky přidání nové osy z přibývá další základní objekt, a tím je rovina. Vektor v prostoru má tři složky a je opět definován dvěma body. Vektorům v prostoru přibývá nová operace vektorový součin. Výsledkem vektorového součinu je nový vektory kolmý na vektory, ze kterých je počítán. Vektorový součin se značí pomocí znaku × a počítá se jako:

$$u \times v = n|u||v| \sin \varphi, \tag{2.23}$$

kde φ je úhel mezi vektory u a v, n je jednotkový vektor kolmý na oba vektory u a v. Máme-li rovinu definovanou třemi body A, B, C, pak je rovina určena dvěma směrovými vektory $\overrightarrow{AB}$ a $\overrightarrow{AC}$, které na ní leží. Obecná rovnice roviny v prostoru je definována:

$$ax + by + cz + d = 0 \tag{2.24}$$

Koeficienty a,b a c odpovídají normálovému vektoru, který lze získat vektorovým součinem směrových vektorů $\overrightarrow{AB}$ a $\overrightarrow{AC}$. Parametr d je potom dopočítán dosazením některého bodu roviny do rovnice.

2.3.3 Polygony

Polygon je v matematice synonymum pro mnohoúhelník. Může být tvořen n vrcholy, nabývat různých tvarů, vytvářet konvexní nebo nekonvexní obrazce, nebo i obsahovat v sobě díry. V počítačové grafice má na rozdíl od mnohoúhelníku polygon jednu důležitou vlastnost, kterou je orientace. V základu se jedná o množinu orientovaných úseček, co dává možnost rozhodování o směru normály polygonu. Orientace může být po směru hodinových ručiček nebo proti směru hodinových ručiček. V počítačové grafice je polygon základní zobrazovací jednotkou, přičemž nejmenší možný polygon je trojúhelník.

Výpočet obsahu konvexního polygonu

Pro výpočet obsahu konvexního polygonu. Z bodů se sestaví matice, jejíž determinant je dvojnásobkem obsahu. Obsah S se pak vypočítá podle následujícího vzorce [3]:

$$S = \frac{1}{2} \sum_{i=0}^{n} (x_{i+1} + x_i)(y_{i+1} - y_i), \tag{2.25}$$

kde body polygonu jsou uspořádány v množině $\{x_0; x_1; \ldots; x_n; x_0\}$ s opakováním prvního prvku na konci množiny. Tato rovnice je určena pro polygony, které jsou uspořádány podél hodinových ručiček. Použijeme-li polygon s opačným směrem, dostaneme stejný, ale záporný výsledek.

Stanovení zda se bod nachází uvnitř polygonu

Pro stanovení zda se bod nachází uvnitř obecného polygonu, se využívá algoritmu založeném na počtu otáčení (angl. winding number). Algoritmus pracuje na zjištění počtu otáčení v námi hledaném bodě vzhledem k polygonu, což je počet otočení, které uděláme okolo bodu, když procházíme postupně po stranách polygonu. Pokud je toto číslo nenulové, bod se nachází uvnitř polygonu.

```
Nastav proměnou omega na 0
Postupně procházej všechny hrany polygonu:
        Pokud daná hrana protíná horizontální osu zkoumaného bodu
        A Pokud je hrana orientovaná doprava
                Uprav číslo otočení omega dle směru
```

Algoritmus 2.3: Algoritmus počítání počtu otáčení v polygonu

Dělení polygonu přímkou

Dělení konvexních polygonů je založeno na algoritmu ořezání polygonu, který popsali ve své práci Greinera a Hormana [4]. Rozdělení polygonu přímkou je zároveň jednodušší a složitější než algoritmus ořezání. Jednodušší protože můžeme ořezávat podle zadané přímky a složitější protože musíme zachovat obě strany ořezaného polygonu. Následující algoritmus provádí dělení.

```
Vytvoř prázdný seznam pro výstupní polygony
Vytvoř prázdný seznam pro návraty
Najdi všechny průniky mezi polygonem a přímkou a seřaď je podél přímky.
Vytvořit páry těchto průniků úsečky
Vytvořit nový prázdný polygon, přidej ho do výstupních polygonů a označ ho jako aktuální
Procházej polygonem a pro každou hranu:
        Přidej počáteční bod do aktuálního polygonu
        Pokud na hraně je průnik s přímkou:
                Přidej bod průniku do aktuálního polygonu
                Najdi bod průniku v párech
                Nastav jeho protějšek jako návrat pro aktuální polygon
                Pokud existuje polygon pro s návratem pro tuto hranu
                        Nastav ten polygon jako aktuální
                Jinak
                        Vytvoř nový polygon do výstupu, a do návratu a nastav ho jako aktuální
                Přidej bod průniku do nově aktuálního polygonu
```

Algoritmus 2.4: Dělení polygonu přímkou

2.4 L-Systémy

L-systémy jsou ve své nejjednodušší podstatě regulárními nebo bezkontextovými gramatikami, na jejichž výzkum mají největší podíl Aristid Lindermayer a Przemyslaw Prusinkiewicz [5]. Díky tomu se v některých literaturách tyto algoritmy namísto L-systémů nazývají Lindenmayerův systém. L-systém je deterministický, ale můžeme se také setkat s L-systémy využívající generátory čísel. Pak hovoříme o stochastickém L-systému. L-Systémy využívají iterace a rekurze. Díky tomu je do výsledného generovaného obrazu zakomponována jistá pravidelnost.

Oproti klasickým gramatikám, které se velmi často využívají v teoretické informatice a oborech zkoumajících jazyky, L-systémy nerozlišují terminální a neterminální symboly. Tyto dvě množiny jsou tedy sloučeny do jediné množiny. L-systém jsou A. Lindermayerem definovány následovně:

Definice 2.1. L-systém G je trojice (V, w, P), kde:

- V je konečná množina symbolů neboli abeceda, V^* je množina všech slov nad abecedou V a V^+ je množina všech neprázdných slov nad abecedou V.
- $w \in V^+$ je neprázdné slovo zvané axiom, které značí počáteční stav L-systému
- $P \subset V \times V^*$ je konečná množina součinů (a, x), která značí přepisovací pravidla

Součin $(a, x) \in P$ se zapisuje $a \to x$, kde $\to$ značí derivaci. Symbol a je levá strana a slovo x je pravá strana přepisovacího pravidla. V množině pravidel P pak musí existovat pro všechna $a \in V$ alespoň jedna dvojice $(a, x), x \in V^*$. L-systém je deterministický právě tehdy, když pro každé $a \in V$ existuje právě jedno přepisovací pravidlo $a \to x$, kde $x \in V^*$.

Definice 2.2. Nechť $\mu = a_1 \dots a_n$ je slovo nad abecedou V. Slovo $v = x_1 \dots x_n \in V^*$ je derivované (nebo generované) z μ, zapisováno $\mu \Rightarrow v$ právě tehdy, když $a_i \to x_i$ pro všechna $i = 1, \dots, n$

Jelikož nelze určit konec generování z důvodu nepřítomnosti terminální, resp. neterminálních symbolů, je generování ukončeno po určitém množství iterací.

Želví grafika

Želví grafiku si lze jednoduše představit jako pohybující se želvu, za kterou zůstává stopa. L-systém pak definuje natočení želvy a vzdálenost, o kterou se má želva pohnout. Želva má následující vlastnosti poloha, orientace (směr pohledu želvy) a případně i štětec, kterým želva zanechává stopu. Ve 2D grafice lze verzi bez štětce zapsat jako trojici (x, y, θ), kde x a y značí souřadnice na kreslícím plátně a θ je úhel natočení proti vodorovné ose.

Příklad 2.1. Zvolíme množinu V, která obsahuje prvky $\{F, +, -\}$, kde F odpovídá posunutí dopředu s vykreslováním stopy, $+$ je otočení o zadaný úhel do kladného směru (po směru hodinových ručiček) a $-$ jako otočení do záporného směru. Množina P obsahuje pouze jediné pravidlo, a to je charakterizováno pomocí $p: F \to F - F + + F - F$, což znamená, že v každé iteraci je znak F nahrazen řetězcem $F - F + + F - F$. Počáteční axiom w je nastaven jako $F + + F + + F$. L-systém tedy vypadá následovně:

L-systém: $G = (V, w, P)$, kde:

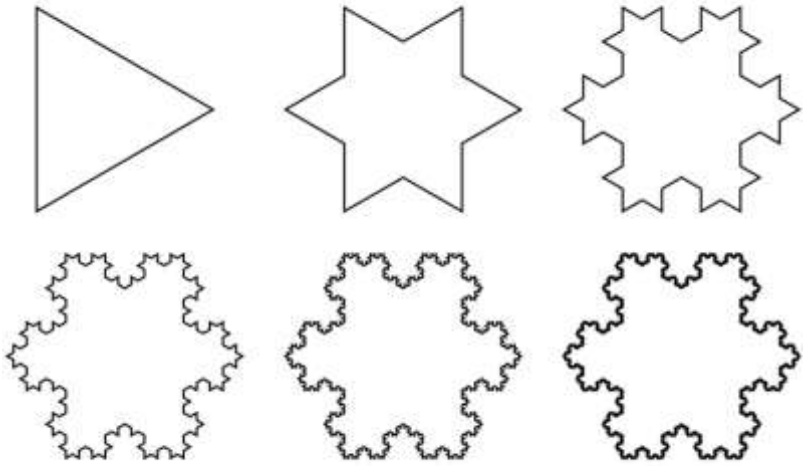
$V = \{F, +, -\}$

$w = F + + F + + F$

$P = \{F \to F - F + + F - F\}$

počáteční úhel želvy = 90°, úhel otočení = 60°, a počet iterací 5

Tímto L-systémem se vykreslí Helge von Kochova sněhová vločka, kterou je možné pozorovat na Obrázek 2.4. Na obrázku je možné vidět postupné iterace od 0 do 5.



Obrázek 2.4: Sněhová vločka Helge von Kocha (Zdroj: [2])

Závorkové L-systémy

Pomocí L-systému zmíněného výše nelze vytvářet větvící se struktury z důvodu, že si želva nepamatuje své předchozí pozice. Tento nedostatek Lindenmayer [5] vyřešil zavedením zásobníku do gramatiky a nad zásobníkem dvě základní operace: uložení stavu želvy na zásobník a vyjmutí posledního uloženého stavu želvy z vrcholu zásobníku. Stavem želvy, jak bylo zmíněno už výše, se rozumí pozice a natočení želvy vůči souřadným osám. Do množiny symbolů jsou přidány dva nové symboly: [pro uložení stavu želvy na zásobník a] pro vyjmutí stavu želvy z vrcholu zásobníku. Podle těchto znaků je metoda pojmenována závorkový L-systém.

Parametrické L-systémy

Parametrické L-systémy jsou označovány jako PL-systémy a rozšiřují L-systém o parametry symbolů. Ke každému symbolu může být přiřazen libovolný konečný počet parametrů. Parametry je obvykle reálné číslo, se kterým se dají provádět běžné matematické operace. Jako parametr může být dosazen i aritmetický výraz, funkce nebo proměnná. Hodnota parametru je pak podle potřeby vyčíslena. Lindenmayer [5] zavedl notaci, že se parametry zapisují do kulatých závorek, které pak nejsou součástí abecedy a jako oddělovač jednotlivých parametrů se používá čárka.

Pokud se vrátíme k pohybu želvy po kreslícím plátně, želva se mohla otáčet pouze o specifikovaný úhel. Díky parametrickým L-systémům můžeme želvě specifikovat velikost úhlu, o který se má otočit.

[2] http://www.ecouterre.com/wp-content/uploads/2012/04/biomimicry-koch-snowflake.jpg

Stochastické L-systémy

Deterministické L-systémy se vyznačují tím, že je jednoznačně definováno jak bude vypadat n-tá iterace algoritmu. Při opakování algoritmu je vždy stejná (viz Příklad 2.1). L-systémy, které v sobě obsahují náhodnost, přičemž geomerický nebo topologický tvar generovaného objekt se nemění, se nazývají stochastické L-systémy. Lindenmayer [5] zavedl náhodnost do implementace L-systému tak, aby gramatika zůstala stejná. Pomocí vygenerovaného pseudonáhodného čísla můžeme měnit délku kroku želvy a úhel jejího otáčení.

Další možnost dosažení stochastického L-systému je přidání nedeterminismu. Toho je docíleno pomocí více pravidel se stejnou levou stranou. Každé pravidlo má potom svoji váhu, která určuje pravděpodobnost vybrání daného pravidla. Pravděpodobnost, že se dané pravidlo vybere, je pak určeno pomocí podílu hodnoty váhy a součtu vah všech pravidel.

Kontextové L-systémy

Kontextový L-systém je velmi podobný kontextové gramatice. Kontextem se rozumí okolní symboly právě přepisovaného symbolu. Oproti bezkontextovým L-systémům jsou zde zavedeny pravidla, které mají více symbolů na levé straně přepisovacího pravidla. Symboly vlevo, resp. vpravo od přepisovaného symbolu se nazývají levý, resp. pravý kontext. Kontextové L-systémy můžeme dělit na kontextové L-systémy, které mají v pravidlech vždy jenom pravý nebo levý kontext, nikoliv však oba naráz. A L-systémy u nichž pravidla mohou obsahovat kontexty oba. Takové L systémy jsou označeny 1L-systémy (pouze jeden kontext) a 2L-systémy. Přepisovací pravidlo 2L-systému pak má tvar $a_l < a > a_p \rightarrow x$, kde a_l je levý kontext a a_p je pravý kontext. Znaky < a > nepatří do abecedy, ale ukazují levý, resp. pravý kontext přepisovacího pravidla. Symbol a je přepisovaný symbol a x je slovo, kterým se přepisovaný symbol přepíše. U 1L-systémů se pak pravý nebo levý kontext do pravidla nepíše, v závislosti na tom, který v daném L-systému neexistuje.

2.5 Tenzorová pole

Tenzor je pojem užívaný ve fyzice, matematice a lineární algebře zobecňující pojmy vektor a skalár. Tenzor, podobně jako vektor, představuje objekt, jehož vlastnosti nezávisí na volbě souřadnic a je vyjádřen pomocí složek. Zatímco u vektoru je možné tyto složky označit jedním indexem, tenzor může nabývat více indexů. To nám určuje řád tenzoru. Každý z indexů se potom pohybuje nad počtem dimenzí použitého prostoru. Speciální případy tenzorů jsou skaláry (tenzor nultého řádu), vektory (tenzor prvního řádu) a matice (tenzor druhého řádu).

Tenzorovým polem se označuje tenzorová veličina, která je definována v každém bodě zkoumaného prostoru. V každém bodě je tedy definován určitý tenzor, jehož řád je v celém

uvažovaném prostoru stejný, ale mění se pouze hodnoty jeho složek. Pole může být definováno na libovolném zkoumaném prostoru.

Chen, Esch a Wonka [6] představily způsob generování silničních sítí na základě vzorů definovaných pomocí tenzorových polí. Výhodou tohoto přístupu je možnost interaktivně navrhnout vzhled města. V práci je dále pojem tenzor označen jako symetrický tenzor druhého řádu na dvojrozměrném prostoru, který je odpovídá této rovnici

$$t = R\begin{pmatrix}\cos 2\theta & \sin 2\theta \\ \sin 2\theta & -\cos 2\theta\end{pmatrix}, \tag{2.26}$$

kde $R \geq 0$ *a* $\theta \in \langle 0; 2\pi)$. Vlastní vektory tenzoru *t* jsou dány rovnicemi

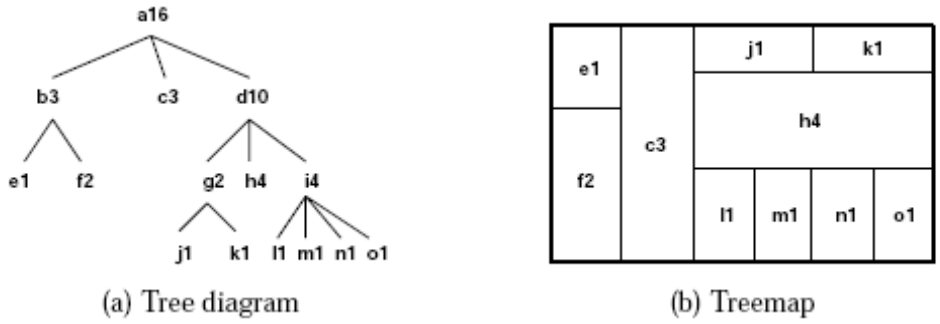
$$eigen_{major} = \lambda\begin{pmatrix}\cos\theta \\ \sin\theta\end{pmatrix}, \tag{2.27}$$

$$eigen_{minor} = \lambda\begin{pmatrix}\cos\theta + \frac{\pi}{2} \\ \sin\theta + \frac{\pi}{2}\end{pmatrix}, \tag{2.28}$$

kde λ je vlastní číslo a platí $\lambda \neq 0$. Tenzorové pole je spojitá funkce, která pro každý bod $p = (x, y) \in \mathbb{R}^2$ definuje tenzor $T(p)$.

2.6 Squarified Treemap

Treemapy jsou datové struktury pro vizualizaci hierarchických objektů. Mezi takové objekty patří například hierarchická struktura adresářů souborového systému. Mají stejnou vyjadřovací schopnost jako stromové struktury a jejich hlavní výhodou je přehledná vizualizace prvků. Zatímco u složitých stromových struktur, které mají mnoho větví není zobrazení příliš přehledné. Srovnání hierarchického stromu a treemapy je možné vidět na Obrázek 2.5.

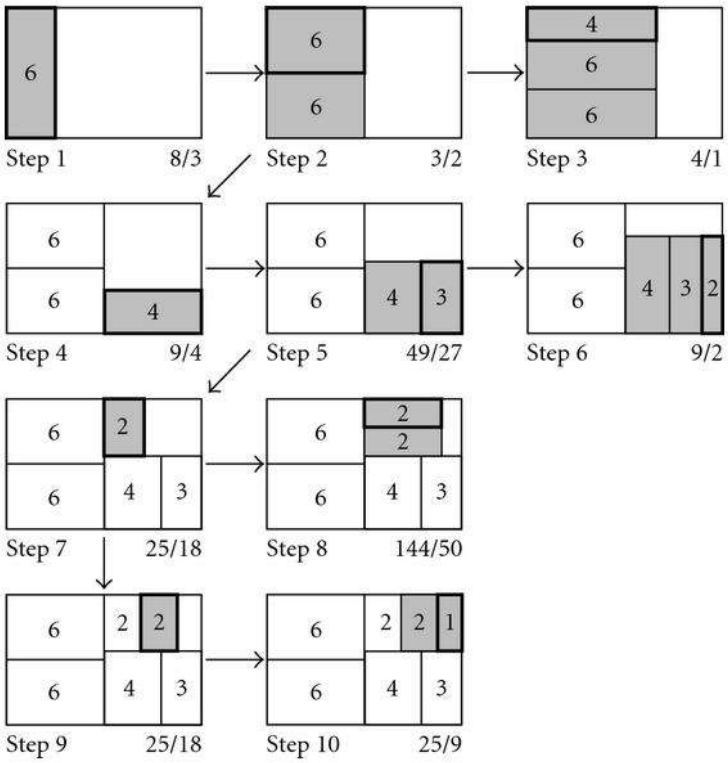


Obrázek 2.5: Zobrazení dat pomocí stromu a treemapy (Zdroj:[7])

Bruls, Huizing a van Wijk [7] predstavili Squarified treemaps algoritmus, který slouží k dělení obdélníku na menší obdélníky za účelem, aby veškeré obdélníky měly co největší poměr stran. Na začátku algoritmu je třeba znát, na kolik částí chceme obdélník rozdělit a také jednotlivé obsahy které dílčí obdélníky mají mít. Nejprve se obsahy dílčích obdélníků se pak seřadí podle velikosti. Dělení

obdélníku probíhá tak, že se původní obdélník rozdělí na dva podle delší z jeho stran. Poté se otestuje zda-li další obdélník v pořadí by poměry stran vylepšil, pokud by se připojil k právě dělenému obdélníku. Pokud tomu tak je tak se oba dělené obdélníky berou jako jeden celek dokud algoritmus neskončí. Jestliže ovšem by poměr stran zhoršil je proces dělení popsaný výše proveden na zbývajícím obdélníku.

Příklad algoritmu squarified treemaps je zobrazen na Obrázek 2.6. Na obrázku je možné pozorovat dělení obdélníku velikosti 6 krát 4 na 7 obdélníků o obsazích 6,6,4,3,2,2,1. U jednotlivých kroků lze pozorovat poměr stran nově přidaného obdélníku, který by měl být co nejmenší.



Obrázek 2.6: Dělení pomocí algoritmu squarified treemap (Zdroj: [7])

3 Existující řešení

V této kapitole jsou představeny různé existující implementace zabývající se procedurálním generování v oblasti měst. U každé z metod se berou v potaz míra realistického modelu a náročnost výpočtu.

3.1 Undiscovered City[8]

Stefan Geuter a další navrhli řešení pro procedurální generování města v reálném čase. Aplikace vytváří silniční systém pomocí jednoduchého mřížkového vzoru, dle kterého se dále generují budovy pomocí kombinací jednoduchých geometrických primitiv.

Použití tohoto systému však nese nevýhody toho, že vygenerovaný systém není příliš realisický. Naopak tento systém je velmi rychlý na tvorbu při generování v reálném čase.

3.2 CityBuilder[9]

Watson ve své práci CityBuilder využil metodu založenou na hraní her pro vygenerování měst. Generování města je implementováno jako simulace města pomocí sady agentů, kteří mohou vytvářet určité části města, například plánovací úřady, dopravní podniky apod. Tento systém nevytváří jenom budovy a silnice ale i simuluje postupný růst města časem.

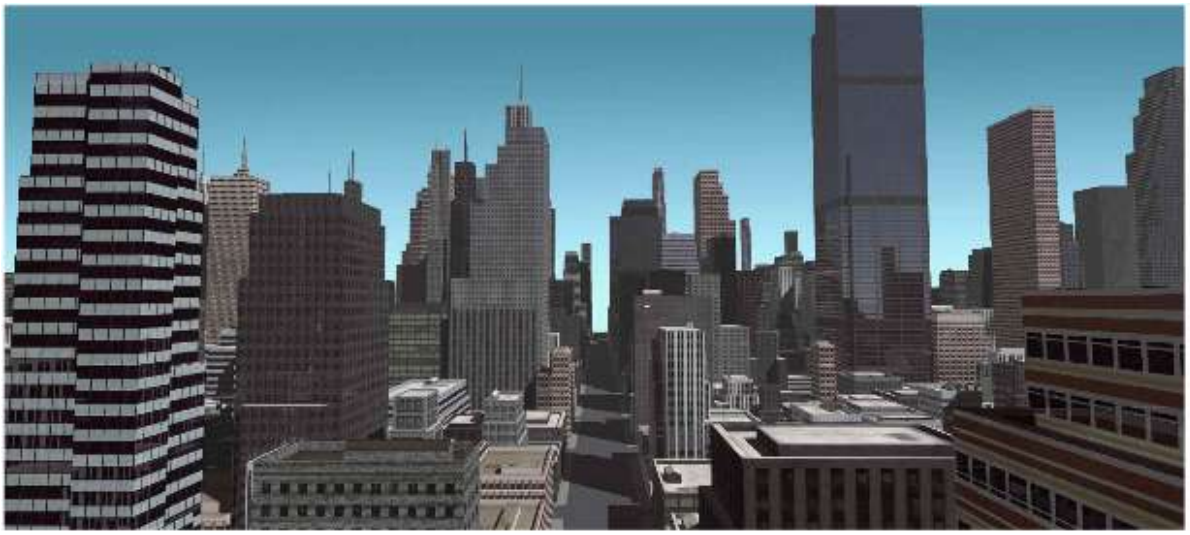
Tento systém je velmi náročný na výpočet protože i malé město se může generovat 15-30 minut. Generované město ale pak může vypadat velmi realisticky, protože byl brán v potaz postupný růst města.

3.3 CityEngine[10]

Autory CityEngine jsou Parish a Müller, který tento systém prezentovali v článku Procedural Modeling of Cities. Jejich systém využívá statistická a geografická data na vstupu generátoru. Pomocí nich se stanoví pozice řek, jezer, populačních center a také silniční vzory, výškové mapy a zóny města. Pro generování silnic využívají parametrický L-systém na němž je specifikováno 9 pravidel. Budovy jsou generovány pomocí stochastických parametrických L-systémů.

Podle zadaných vstupních map metoda přináší velmi realistické výsledky a i výpočetní náročnost algoritmů není příliš veliká.

Z této práce bylo čerpáno při návrhu algoritmu pro tvorbu silničního systému.



Obrázek 3.1: Ukázka z CityEngine (Zdroj: [10])

4 Použité knihovny a aplikační rozhraní

Tato kapitola se zabývá popisem jednotlivých nástrojů, které jsou použity v implementaci výsledného demonstračního programu. Jako základ byl použit jazyk C++, jednak z důvodu výkonu ale také přirozeného propojení ostatních použitých knihoven, které jsou rovněž implementovány v jazyce C/C++.

Pro vytvoření okna a získání uživatelského vstupu je použita knihovna GLFW. O samotné zobrazení se stará grafická OpenGL na verzi 3.3. Pro práci s vektory a maticemi byla využita matematická knihovna GLM.

4.1 GLFW

GLFW je multiplatformní open-source knihovna pro vývoj programů využívajících OpenGL. Knihovna obsahuje podporu pro obsluhu vstupních zařízení, jako jsou klávesnice, myš, joystick a dalších. GLFW vytváří okno a poskytuje OpenGL kontext na něj. Tento kontext si lze zjednodušeně představit jako plátno, se kterým OpenGL dokáže pracovat a vykreslovat do něj.

V této práci je knihovna GLFW využita pro vytvoření okna programu, získání vstupu ze vstupních zařízení a díky tomu i k řízení programu. Pro tento účel existuje mnoho dalších aplikačních rozhraní, jmenovitě například SDL, freeGLUT. Hlavními důvody výberu knihovny GLFW oproti ostatním se staly:

- dostupná a kvalitní dokumentace
- jednoduché vytvoření okna i OpenGL kontextu

4.2 OpenGL

OpenGL (Open Graphics Library) je nízkoúrovňová knihovna pro práci s trojrozměrnou grafikou. Od doby svého uvedení na počátku devadesátých let se stala široce uznávaným a podporovaným standardem na poli grafických aplikací, CAD/CAM/CAE inženýrských systémů, virtuální reality a tvorby her. OpenGL představuje jednotné API (Application Programming Interface), mezi programem a grafickým hardware. Hlavním rysem OpenGL je nezávislost na cílové platformě a použitém programovacím jazyce.

OpenGL podporuje široké spektrum programovacích jazyků a různých platforem. Tradičními platformami jsou Windows, Linux a Mac OS. Navíc OpenGL ES přináší podporu i pro mobilní a vestavěná zařízení. V této práci je OpenGL využito pro vykreslování výsledné scény města do okna programu.

4.2.1 OpenGL shadery

Pro účely práce jsou též využity jednoduché shadery. Tyto programovatelné části grafické renderovací pipeline vznikly jako nástupcem ARB assembly language, který byl příliš komplexní a neintuitivní. Implementační jazyk shaderů se nazývá OpenGL Shader Language (zkráceně GLSL) a je založený na syntaxi jazyka C. Hlavním důvodem jejich využití je větší flexibilita při programovaní pipeline, která do té doby byla fixní. Jejich využití je především v herním průmyslu pro tvorbu různých grafických efektů.

Vertex Shader

Je první programovatelnou složkou grafické pipeline. Pracuje na úrovni jednotlivých bodů, ze kterých je zpracovávaný model složen. Ve vertex shaderu jsou řešeny různé transformace bodů modelu, aniž by se zasahovalo do originálních dat. Tímto způsobem je možné docílit jednoduchých animací a transformování tvaru objektu.

Geometry Shader

Tento shader se v grafické pipeline objevil až ve verzi OpenGL 3.2. Jeho účelem jsou transformace na úrovni jednotlivých grafických primitiv (body, přímky, polygony). Na základě vstupních primitiv lze navíc vytvářet nové body a primitiva. Tímto způsobem lze změnit kompletní tvar vstupního objektu. Vstupem může být například jediný bod a za pomocí odchylek od daného bodu lze vytvořit například krychli.

Fragment Shader

Po skončení rasterizace všech primitiv je další částí grafické pipeline Fragment shader. Fragment shader na vstup dostává jednotlivé pixely po rasterizaci. Má tak možnost pracovat tak s barevnou složkou pixelů, což znamená i nastavení barvy podle zadané textury. Zde se vytváří například různé barevné filtry.

4.3 GLM

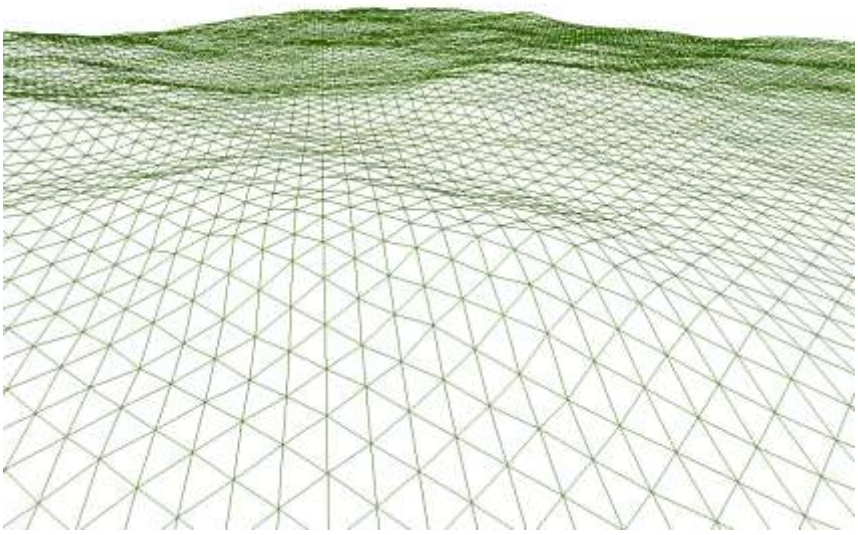
OpenGL Mathematics (GLM) je multiplatformní matematická knihovna pro jazyk C++ založená na GLSL specifikacích. GLM poskytuje třídy a funkce které jsou navrženy, implementovány a pojmenovány stejně jako jejich protějšky v GLSL. GLM nabízí oproti základním funkcím GLSL navíc i další funkce jako jsou: transformace matic, kvaterniony, náhodná čísla, šum, atd. Knihovna byla vybrána právě z tohoto důvodu.

5 Návrh a implementace

Generování města je rozděleno do několika provázaných částí, které jsou v následujících podkapitolách dále rozvinuty. Nejdříve je vytvořen terén, který slouží jako základ ostatních generátorů. Následuje generátor silniční sítě, který podle stanoveného vzoru z nastavení programu vytvoří silniční síť, jež definuje bloky, které jsou pak rozděleny na parcely. Na parcelách se dále vytvoří budova včetně interiérů. Každá z těchto částí produkuje modely, které se nakonec zobrazují ve scéně.

5.1 Terén

Základem vytvoření terénu je výšková mapa. Výšková mapa je složena z pravidelné mřížky, jejíž indexy jsou souřadnice v prostoru a hodnota je výška v daném bodě. Pro získání výšky bylo využito Perlinova šumu s nastavením aby vytvářel mírně kopcovitý terén. Vygenerovaný terén je možné sledovat na Obrázek 5.1.



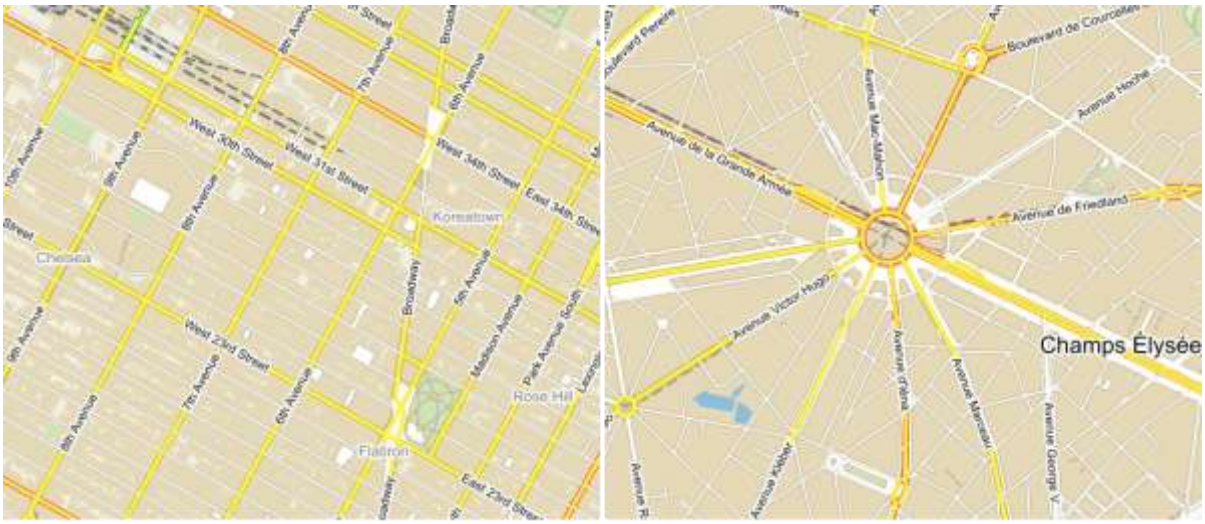
Obrázek 5.1: Drátěný model vygenerovaného terénu.

5.2 Silniční systém

Silniční sítě jsou klíčovým aspektem města. Počet silnic a jejich uspořádání je dáno historickým vývojem města. Silniční sítě je obtížné zobecnit, protože jednotlivé silnice jsou mezi sebou různě

propojeny a jsou součástí velmi složitého systému. Při pohledu na mapy reálných měst lze pozorovat několik vzorů silničních sítí. Právě tyto vzory jsou klíčovým aspektem k utvoření procedurálního generátoru, protože zahrnují strukturu silniční sítě. Existuje značné množství vzorů, které jsou ve městech použity. V rámci tohoto projektu jsou implementovány následující silniční vzory:

- Mřížkový vzor - jednotlivé městské bloky mají tvar obdélníku a silnice, které je obklopují, svírají mezi sebou pravý úhel. Tento vzor je typický pro moderní části měst. Obrázek 5.2 vlevo.
- Kruhový vzor - formuje hlavní silnice v soustředných kruzích. Dále obsahuje silnice vycházející ze středu. Nejčastěji se vyskytuje v okolí nějakého památníku. Obrázek 5.2 vpravo.



Obrázek 5.2: Vzory silničních sítí na reálných mapách (Zdroj:[3])

Tento generátor má dva hlavní úkoly. Vytvoření silniční sítě, definování bloků mezi silnicemi a rozdělení těchto bloků na jednotlivé parcely. V následujících podkapitolách jsou popsány detaily jejich práce.

5.2.1 Generování silniční sítě

První část je založena na růstovém algoritmu. Hlavní funkce tohoto algoritmu je řízena pomocí funkcí GlobalGoals (dále jen globální cíle) a LocalConstraints (dále jen lokální podmínky). Globální cíle navrhují další segment silničního systému, pomocí kterého se silniční systém dále rozroste. Navržený segment, může být například určen dle nějakého vzoru nebo v závislosti na vzdálenosti od centra města. Další skutečností je, že tento navržený segment je generován bez ohledu za je uskutečnitelný. Ke kontrole a případné úpravě takového segmentu slouží lokální podmínky. Hlavním jejich úkolem je zkontrolování generovaného segmentu zda odpovídá veškerým stanoveným podmínkám blízkého

[3] https://mapy.cz

okolí segmentu a provedení nezbytných operací, aby tyto podmínky vyhověli. Není-li možné segment takto upravit je zahozen. Základní princip tohoto algoritmu lze sledovat na následujícím pseudokódu:

```
Přidání počátečního bodu do fronty Q
while not Empty(Q):
        segment = Pop(Q)
        accepted, modified =  LocalConstraints(segment)
        Pokud accepted je nepravda pokračuj od začátku cyklu

        Add(Edges,modified)
        Pro každý segment sn navržený GlobalGoals(segment)
                Add(Q,priority(sn), sn)
end while
```

Algoritmus 5.1: Algoritmus pro generování silniční sítě

Algoritmus vyžaduje prioritní frontu Q, na které jsou definovány operace zjištění zda je fronta prázdná (Empty(Q)), vybrání prvku s nejvyšší prioritou z fronty(Pop(Q) a přidání prvku včetně jeho priority do fronty(Add(Q,priority,item)). Tato fronta slouží pro uchování další bodů pro další expanzi. Dále je zapotřebí GlobalGoals a LocalConstraints, jejichž principy jsou popsány výše a budou ještě dále rozvinuty. A je také zapotřebí kolekce Edges, která uchovává přijaté segmenty silniční sítě.

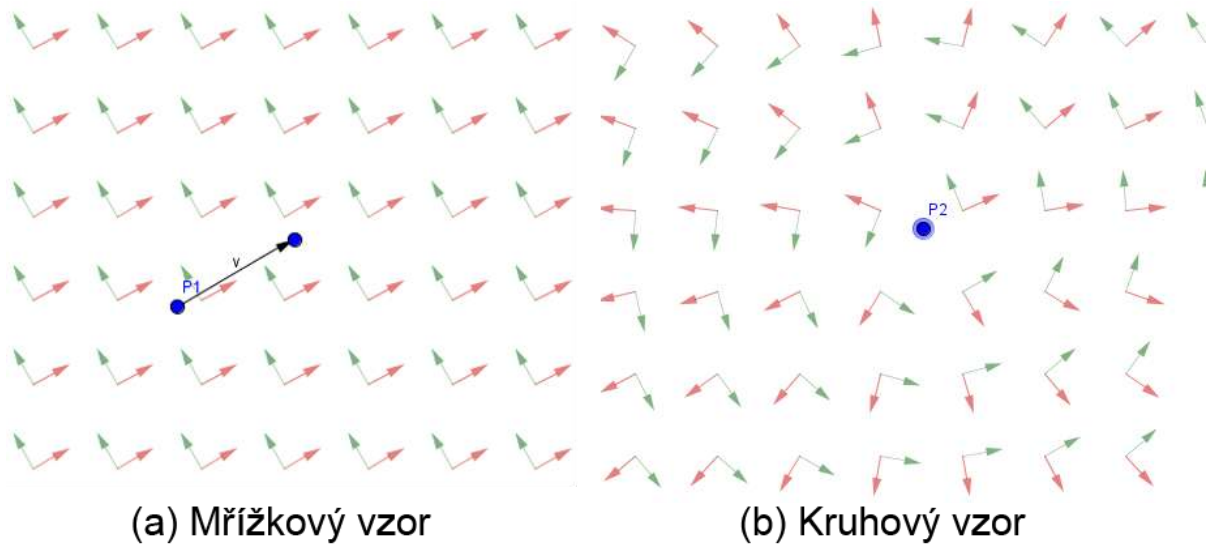
Jednotlivé segmenty jsou určeny tak, aby bylo možné pro celou kolekci získat jednotlivé bloky, které jsou svírány mezi silnicemi. Tyto bloky se pak dále dělí na jednotlivé parcely, které slouží dále pro vytváření budov.

Výstupem této části generátoru města je kolekce vrcholů, která zároveň s pozicí obsahuje i kolekci referencí na jednotlivé segmenty které v daném bodě začínají. Dále kolekce segmentů, která má stanovený počáteční vrchol a koncový vrchol. A nakonec obsahuje i vygenerované parcely, které obsahují kolekci vrcholů, jež parcelu ohraničují a kolekci hran této parcely které jsou přístupné ze silnice.

Globální cíle

Nejdříve jsou dle zadané konfigurace vygenerována tenzorová pole. Tato tenzorová pole slouží jako realizace silničních vzorů, které jsou popsány v kapitole 5.2Silniční systém. Pro implementaci jsou vybrána tenzorová pole, protože díky nim lze jednoduše kombinovat různé silniční vzory. Pro získání směru pokračování silnice jsou z tenzorových polí stanovena vektorová pole. Tato vektorová pole odpovídají vlastním vektorům jednotlivých tenzorů, které jsou získány na základě tenzorových polí. Pro tenzory druhého řádu, které silniční systém využívá, jsou tyto vektorová pole právě dvě. Pro další reference si tato pole označíme jako hlavní vektorové pole a vedlejší vektorové pole. Jelikož nám stačí znát pouze směr, jsou vektory uvnitř těchto polí jednotkové a při generování se prochází oběma směry. Jednotlivé zobrazení tenzorových polí a směry jejich vlastních vektorů lze pozorovat na Obrázek 5.3. Pro přehlednost je zobrazen pouze jeden směr vektorových polí získaných z

tenzorových polí. Červenými šipkami je vyznačeno hlavní vektorové pole a zelenými vedlejší vektorová pole. Dále je na Obrázek 5.3.a možné vidět počáteční bod P1 a směrový vektor v který definuje mřížkový vzor. Naopak kruhový vzor na Obrázek 5.3.b je definován pouze pomocí středu P2.



Obrázek 5.3: Vlastní vektorová pole získaná z tenzorových polí.

Pro získání segmentu určité délky je potom zapotřebí postupně procházet zvoleným vektorovým polem dokud nedosáhneme požadované délky. Toho je docíleno pomocí stopování, následující algoritmu popsaného pomocí pseudokódu:

```
endPoint = start
dokud je délka mezi start a endpoint menší než stanovená délka nebo nebyl překročen počet iterací:
        direction = sample_vector_field(endPoint)
        next = endPoint + direction
        Pokud rozdíl úhlů mezi start a endpoint a endpoint a next je větší než práh
                skonči cyklus
        endPoint = next
end
```

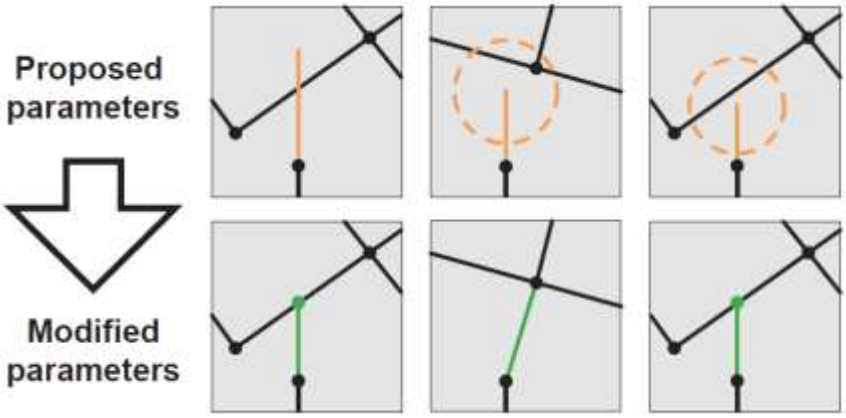
Algoritmus 5.2: Stopování bodu vektorovým polem

Start nám stanovuje počáteční bod, ze kterého začalo stopování a endPoint nám na konci algoritmu bude značit koncový bod. Sample_vector_field(position) nám vrátí vektor z vektorového pole na pozici position. Pro plynulejší přechody mezi jednotlivými body může být sample_vector_field implementována pomocí numerické integrace. Dále může nastat, že celková vypočítaná délka pomocí tohoto algoritmu nebude odpovídat naši požadované délce díky tomu, že cesta vektorovým polem je příliš zakřivená.

Lokální podmínky

Lokální podmínky slouží k porovnání segmentu k jeho nejbližšímu okolí a stanovení zda-li je segment přijat, zahozen anebo upraven pomocí některých z pravidel. Podmínky implementované v této práci jsou následovní (viz Obrázek 5.4):

- Pokud se segmenty navzájem kříží, je generovaný segment zkrácen a druhý segment rozdělen. Díky tomu je vytvořena křižovatka.
- Pokud je koncový bod generovaného segmentu poblíž jiného segmentu, pak je generovaný segment prodloužen a na místě jejich průniku vytvořena křižovatka
- Pokud je koncový bod generovaného segment poblíž jiné křižovatky je prodloužen do ní.



Obrázek 5.4: Kontrola podmínek a případná úprava silničních segmentů (Zdroj: [6],[10])

Rozšíření algoritmu

Na základě předchozích stanovených Globálních cílů a Lokálních podmínek je algoritmus Algoritmus 5.1 rozšířen tak že nevytváří postupně jednotlivé segmenty, ale jsou vytvořeny cesty. Tyto cesty se skládají ze segmentů navržených Globálními cíly a jejich úpravou lokálními podmínkami. Cesty pokračují od zadaného bodu, dokud není splněna některá z těchto podmínek:

- Poslední generovaný segment je příliš krátký.
- Poslední generovaný segment přesahuje přes hranu mapy.
- Poslední generovaný segment končí poblíž již existujícího vrcholu silniční sítě.
- Poslední generovaný segment kříží jiný segment.
    - Poslední generovaný segment tvoří na cestě smyčku. Tedy konec tohoto segmentu je připojen k některému z předcházejících vrcholů cesty.

5.2.2 Generování bloků a parcel

Z předchozí části generátoru máme vytvořenu silniční síť. Ta je složena z vrcholů a hranami mezi nimi. Tato data jsou procházena pomocí algoritmu popsaného dále. Algoritmus vychází z předpokladu, že jeden silniční segment může být připojen pouze ke dvěma blokům. Algoritmu je možné pozorovat na následujícím pseudokódu:

```
Vytvoř kolekci N a přiřaď do něj všechny silniční segmenty
Vytvoř prázdnou kolekci H a která bude obsahovat páry hodnot silniční segment a směr
Vytvoř kolekci B pro uchování bloků
while not Empty(N) && not Empty(H):
        Vytvoř proměnnou start pro uchování startovacího segmentu
        Vytvoř proměnnou dir pro uchování směru
        If not Empty(H):
                start = First(H).segment
                dir = First(H).direction
        ElseIf not Empty(N):
                start = First(N)
                dir = podél
        end If

        Vytvoř kolekci P pro uchování bodů bloku
        Vytvoř proměnnou t pro uchování právě procházeného segmentu a nastav jej na start
        Vytvoř proměnnou startDir pro uchování směru a nastav jej na dir
        do:
                Pokud se segment t nachází v kolekci N:
                        Odstraň segment t z kolekce N
                        Přidej do kolekce H pár hodnot t a směr opačný k dir
                Jinak pokud se pár hodnot t a dir nachází v kolekci H:
                        Odstraň pár hodnot t a dir z kolekce H
                Dle směru dir urči bod point
                Add(P,point)
                t,dir = moveNextSegment(t,dir)
        while ( t != start || startDir != dir)

        Odstraň slepé cesty z P
        Pokud blok definovaný pomocí bodů P je validní:
                Add(B,P)
end while
```

Algoritmus 5.3: Algoritmus pro generování bloků

Algoritmus postupně dvakrát prochází všechny hrany, dokud nejsou obě kolekce N a H prázdné. Odstranění slepých větví bloky probíhá postupným průchodem kolekce bodů P, dokud neplatí následující podmínky:

- Pokud předchozí bod je roven následujícímu bodu.
- Pokud procházený bod je roven následujícímu bodu.

Dále je za potřebí funkce moveNextSegment, která ze stanoveného segmentu a směru navrátí další segment a jeho směr, tak aby byla zachována návaznost společného vrcholu. Společný vrchol je podle směru určen jako počáteční nebo koncový vrchol zpracovávaného segmentu. Dle odkazů tohoto vrcholu na ostatní segmenty je určen následující segment. Může nastat jeden ze tří případů:

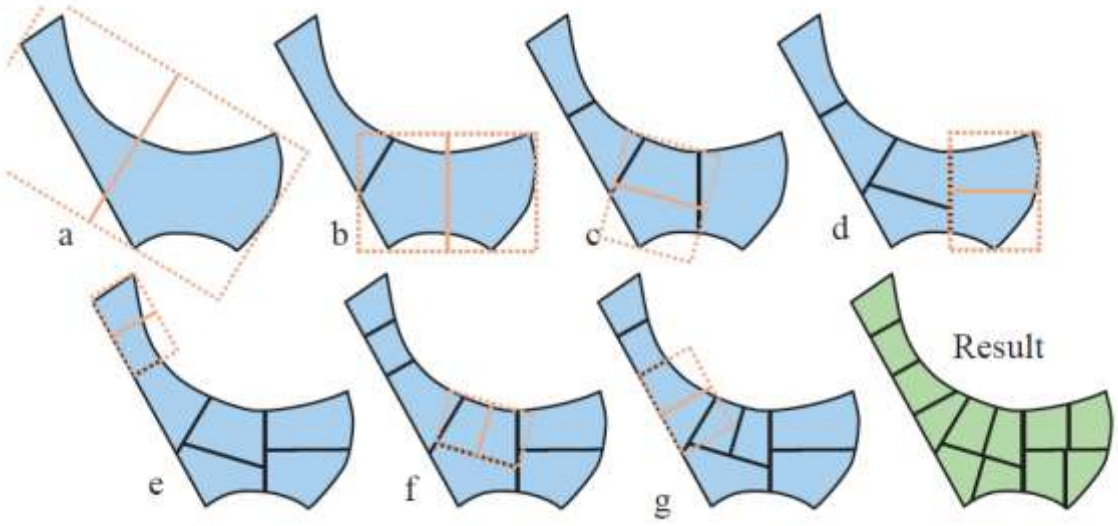
- Vrchol má pouze jednu referenci. To značí, že daná cesta je slepá. Navrátí se stejný segment pouze s opačným směrem.
- Vrchol má dvě reference. To značí rovnou trasu. Navrátí se druhý segment a směr v závislosti na tom, který z bodů segmentu je vrchol.

- Vrchol má tři a více referencí. To značí křižovatku. Navrátí se takový segment z jeho referencí, který má nejmenší úhel svíraný mezi zpracovávaným segmentem touto funkcí a procházeným segmentem.

Výsledkem tohoto algoritmu je kolekce bloků. Body těchto bloků pak představují polygony, které daný blok definují. Většina těchto polygonů je uspořádána podél směru hodinových ručiček. Avšak při generování vzniknou i bloky, které mají body polygonu uspořádány proti směru hodinových ručiček. Tyto polygony pak definují blok, který zahrnují některé ostatní polygony.

Generování parcel

Z předchozího kroku je známa kolekce bloků. Pro generování parcel jsou vybrány pouze bloky, jejichž polygony jsou uspořádány podél směru hodinových ručiček. Na parcele je nalezen takový bounding box, který má ze všech nejmenší obsah ohraničující oblasti. Toho je docíleno pomocí rotace objektu a poté určení bounding boxu zarovnaného na osy. Dále dělení na parcely probíhá rozdělením získaného bounding boxu na poloviny podle kratší z jeho os. K tomu je použit algoritmus popsaný v kapitole 2.3.3. Pro všechny vytvořené polygony tímto dělením jsou zkontrolovány podmínky. Tyto podmínky jsou minimální obsah oblasti, poměr stran a minimálně jedna z hran polygonu musí být přístupná ze silnice. Pokud některý z dílčích polygonů nesplní některou z podmínek pak je výsledná parcela rovna polygonu před dělením. Jinak pokud všechny dílčí polygony splnili všechny podmínky, pak je algoritmus opakován pro každý z dílčích polygonů. Po provedení tohoto algoritmu na všechny bloky je vytvořena kolekce parcel.

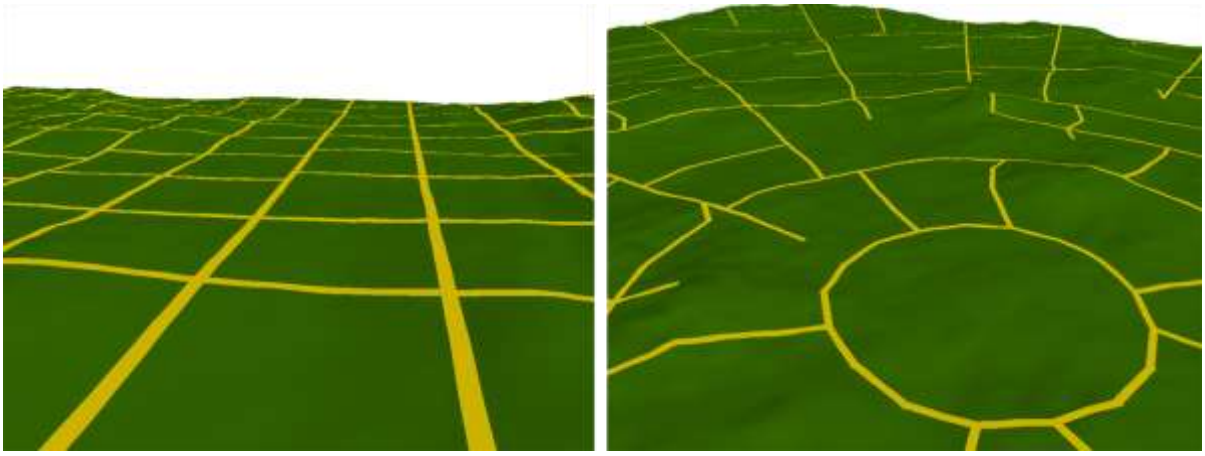


Obrázek 5.5: Dělení pomocí orientovaného bounding boxu (Zdroj: [11])

5.2.3 Generování modelu

Předchozí zpracované kroky vytváří kolekci segmentů, které obsahují vrcholy v 2D prostoru. Jelikož terén může být ve scéně zvlněný a vrcholy jsou umístěny v rovině, je zapotřebí dopočítat třetí

souřadnice vrcholů. Terén je ve scéně zobrazen jako souvislá mřížka s polygony proto je potřeba jednotlivé silniční segmenty podle terénu navzorkovat. Pro segmenty silnic jsou pak modely generovány následovně. Silniční segmenty jsou postupně rozděleny podle vzorkovací frekvence na několik dílčích segmentů. Tyto segmenty jsou pak rozšířeny na obdélníky za pomocí parametru šířky silnice. A pro každý vrchol těchto obdélníku je stanovena výška pomocí jejich souřadnic. Jelikož při vzorkování by mohlo nastat, že by silnice kolidovala s terénem, jsou výšky silničních segmentů posunuty pomocí konstanty výše. Tyto vrcholy obdélníků jsou pak postupně přidány do společné kolekce bodů, které OpenGL renderuje ve scéně. Výsledný vygenerovaný 3D model silnic je možné vidět na následujícím obrázku:



Obrázek 5.6: Vygenerované modely silnic podle vzoru mřížky (vlevo) a podle vzoru radial(vpravo)

5.3 Budovy s interiéry

Generátor budov je inspirován prací Marsona a dalších [12]. Generování budov probíhá v několika krocích. Nejdříve je na parcele určen půdorys, na kterém bude stát budova. Dále je tento půdorys rozdělen na menší části, které značí jednotlivé místnosti. Pomocí místností jsou dále vytvořeny stěny, které místnosti oddělují. Poté je stanovena vstupní místnost a je dopočítáno propojení všech místností tak, aby bylo možné se z každé místnosti dostat do vstupní místnosti. Nakonec je vytvořen 3D model budovy. V následujících podkapitolách jsou dále tyto kroky popsány detailněji.

Vytvoření základního půdorysu

Na vytvoření půdorysu je zapotřebí znát polygon parcely a informace o každé jeho hraně jestli se nachází při silnici nebo ne. Postupně se zpracují všechny hrany nacházející se poblíž silnice a vybere se z nich nejdelší. Pokud jsou dvě hrany vedle sebe téměř rovnoběžné, utvoří dohromady jedinou hranu. Když máme nejdelší hranu tak tato hrana pro nás bude vstupní stěnou do budovy. Dále je zapotřebí dopočítat zbytek půdorysu. Toho je docíleno podle následujícího algoritmu, který se vepisuje co největší obdélník do oblasti polygonu parcely, který je zarovnaný na střed stanovené vstupní stěny. Nejprve jsou zjištěny vektory wDir a lDir. Vektor wDir je jednotkový vektor směru

vstupní stěny a vektor lDir je jednotkový vektor na něj kolmý. Vektor lDir směřuje dovnitř polygonu parcely. Dále je potřeba vědět střed vstupní stěny označený jako center. Funkce calcMinRatio(n1,n2) vypočítá nejmenší poměr mezi zadanými parametry n1 a n2. Pro zachování určitého poměru stran je zde parametr minRatio, který může nabývat hodnot od 0 do 1, přičemž 1 značí poměr stran čtverce a 0 značí obdélník, jehož jedna strana se limitně blíží k 0. Aby nebyl generován příliš malý půdorys je stanovena minimální šířka budovy parametrem minWidth. Pro krokování slouží parametr step, který slouží pro výpočet úbytku mezi jednotlivými iteracemi. Poté je třeba znát délku obalujícího bounding boxu označenou jako bounding_box_length. Výstupem algoritmu jsou bestWidth,bestLength, které značí největší šířku a délku vepsaného obdélníku do polygonu parcely. Pokud takový obdélník nebyl nalezen, zůstávají tyto parametry rovny 0 a je přerušeno další generování budovy. Výsledný obdélník slouží jako půdorys pro další dělení popsáno pomocí následujícího algoritmu:

```
bestWidth=0
bestLenght=0
bestArea=0
Nastav width = Délka vstupní stěny
while(width>minWidth){
        v1 = center - wDir*(width/2);
        v2 = center + wDir*(width/2);
        Pokud body v1,v2 neleží v polygonu parcely:
                Nastav width = width - step a pokračuj od začátku cyklu
        Nastav length = min(bounding_box_length,width/minRatio)
        while( (calcMinRatio(width,length) >= minRatio) &&
                (width * length) > bestArea):
                v3 = v1 + lDir * length
                v4 = v2 + lDir * length
                Pokud body v3,v4 neleží v polygonu parcely:
                        Nastav length = length - step a pokračuj od začátku vnořeného cyklu
                bestArea = width * length,
                bestWidth = width,
                bestLength = length
                A vynoř se z cyklu
        end while
        Nastav width = width - step
end while
```

Algoritmus 5.4: Algoritmus pro stanovení půdorysu na parcele

Generování rozložení místností

Dle stanoveného půdorysu z předchozího kroku se vygenerují místnosti. Algoritmus pro dělení místností využívá metodu squarified treemaps popsanou v kapitole DOPLNIT KAPITOLU. Dělení místností probíhá pomocí dvou kroků. Rozdělení půdorysu do tří obsahově podobných oblastí a jejich dalšího rozdělení. Nejprve se vytvoří kolekce náhodných čísel s četnosti tří prvků. Pro tuto kolekci je zvolen generátor čísel s normálovým rozložením pravděpodobnosti, aby se čísla od sebe příliš nelišila. Poté jsou všechna vygenerovaná čísla sečtena a výsledný součet je označen jako suma. Dále jsou z půdorysu zjištěny parametry šířka, výška a díky nim vypočítán i obsah. Každé vygenerované

číslo je pak vynásobeno podílem $\frac{obsah}{suma}$. Tím jsou stanoveny obsahy oblastí. Tato upravená kolekce je společně se šířkou a výškou vstupem squarified treemap algoritmu, na jehož výstupu jsou obdélníky reprezentující oblasti. Na každé oblasti je poté postup popsaný výše proveden znovu s rozdílem, že pro generování náhodných čísel v kolekci je použito rovnoměrné rozložení pravděpodobnosti a počet dělených místnosti je dán vzorcem dán vzorcem

$$num\_rooms = \lfloor \max\ (\sqrt[3]{\text{area}} - 1{,}1) \rfloor, \tag{5.1}$$

kde area je plocha právě dělené oblasti. Max(par1, par2, ...,parn) značí funkci, která ze zadaných parametrů vybere největší z nich a num_rooms je počet místností na které oblast bude dále dělit. Celý tento výpočet je zaokrouhlen dolů na nejbližší celé číslo.

Tento přístup rozdělování místnosti nadvakrát má výhodu toho, že se velké místnosti negenerují pouze na jedné straně budovy, ale jsou více rozprostřeny. Porovnání přístupu dělení jednou i nadvakrát lze pozorovat na obrázku DOPLNIT OBR OZKAZ. Na obrázku OBR je vyznačený půdorys, ze kterého se provádí dělení.

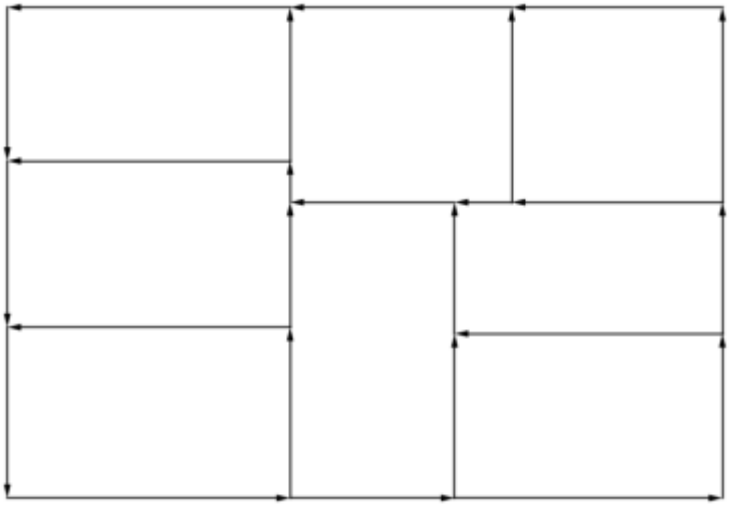
Vytvoření stěn

Poté co jsou známy pozice a rozměry jednotlivých místností se začnou generovat stěny. Každá ze stěn má začátek a konec. Stěny obsahují informace o místnostech, které oddělují a místnosti mají kolekci stěn, které danou místnost definují. Při vytváření stěn jsou stěny testovány zda-li se mezi sebou nepřekrývají. Mohou nastat následující překrytí stěn:

- jedna ze stěn leží uvnitř druhé a nemají žádný společný vrchol
- jedna ze stěn leží uvnitř druhé a mají jeden společný vrchol
- stěny jsou částečně překryty, ale nemají žádné společné vrcholy
- stěny mají oba vrcholy společné

Pokud jsou obě stěny totožné nezávisle na jejich orientaci je tato stěna ponechána a je jí přiřazena další místnost. V závislosti na poloze stěny z kolekce pak dále probíhá dělení. Leží-li generovaná stěna uvnitř zadané stěny je provedeno dělení, přičemž střední části jsou pouze upraveny vrcholy a vytvořeny jedna až dvě nové stěny pokud jsou některé vrcholy sdíleny. Naopak leží-li stěna z kolekce zcela v generované stěně, je stěna z kolekce ponechána a dle její orientace jsou vytvořeny jedna až dvě nové stěny v závislosti na společných vrcholech. Pokud jsou mezi sebou stěny částečně překryty tak se dále rozdělí takovým způsobem, aby na ně šlo použít přecházející kroky.

Po průchodu všech stěn se místnostem přiřadí sousedící místnosti, a to podle místností, které dané stěny oddělují. Na konci těch operací víme, která místnost sousedí s jakou díky stěnám, které je oddělují a stěny. Stěny, jež mají pouze přiřazenou jednu místnost jsou venkovní. Výstup tohoto kroku lze pozorovat na Obrázek 5.7. Stěny jsou zde vyznačeny jako vektory, aby bylo možné pozorovat jejich směr.



Obrázek 5.7: Výsledné rozdělení místností

Stanovení vstupní místnosti

Z předchozích kroků víme, které stěny jsou venkovní a z generování půdorysu známe které z těchto stěn na vstupní straně budovy. Vstupní místnost se určuje dvěmi podmínkami. Stěna vedoucí do vstupní místnosti musí být dostatečně velká, aby bylo možné do ní vměstnat dveře. Šířka dveří je určena podle použité konfigurace. A druhou podmínkou je pokud je budova vícepatrová je vstupní místnost vybrána tak, aby bylo možné do ní vložit schodiště. Rozměry typů schodišť jsou dopočítány podle velikosti patra budovy a pomocí parametrů stanovených v použité konfiguraci aplikace. Jakmile jsou vstupní místnost a stěna vybrány jsou uprostřed této stěny vytvořeny dveře. Tyto dveře se generují pouze na prvním patře. Pokud je budova vícepatrová jsou ve vyšších patrech namísto dveří vytvářeny okna. Nakonec jsou procházeny všechny venkovní stěny a náhodně jsou některé z nich zvoleny a vytvořeny na nich okna.

Vytvoření propojení místností

Dveře jsou v patře umístěny tak, aby bylo možné se z každé místnosti dostat do vstupní místnosti. Vytváření propojení a stanovení délky je založeno na algoritmu prohledávání do šířky (BFS). Pro tento algoritmus je zapotřebí FIFO frontu Q s operacemi: přidání prvku na konec fronty (Enqueue(item,Q)), odebrání prvku ze začátku fronty (Dequeue(Q)) a zjištění zda je fronta prázdná (Empty(Q)). Ke zjištění zda je možné mezi místnostmi vytvořit dveře je za potřebí funkce maxWallLenght(room1,room2), která vrací největší délku stěny, která je sdílená místnostmi room1 i room2. A doorWidth který značí světlou šířku dveří zadané konfigurace. Dále je třeba znát pro každou místnost její sousedy a vzdálenost ke vstupní místnosti. Vzdálenost je brána jako počet místnosti, přes které je nutné projít, aby se z dané místností dalo dostat do vstupní místnosti. Tento parametr je v algoritmu označen jako path. Pokud má path hodnotu -1, ještě nebyla místnost zpracována. Pseudokód algoritmu je popsán následovně:

```
Všechny místnosti: path=-1
Vstupní místnost: path=0
Vlož do fronty Q všechny sousedy vstupní místnosti
while (!Empty(Q)){
        room = Dequeue(Q);
        Pro všechny sousedy neighbour místnosti room,
                které mají path==-1 a nenachází se ve frontě Q:
                Enqueue(Q,neighbour);
        Vyber ze všech sousedů místnosti room souseda s nejmenší path>=0
                a označ jej best
        Má-li více sousedů stejný path, vyber toho, který má nejmenší obsah místnosti
        if(maxWallLenght(room,best) < doorWidth){
                Enqueue(Q,room)
        } else {
                Vytvoř mezi místnostmi room a best dveře
                room.path = best.path + 1
        }
}
```

Algoritmus 5.5: Algoritmus pro propojení místností

Vytvoření modelu

Výsledné vytvoření modelu budovy se skládá ze dvou částí: vytvoření modelů stěn a vytvoření podlah a podezdívky. Z předchozích kroků jsou vytvořeny stěny jako úsečky mezi místnostmi. Je dále potřeba stěnu rozšířit o tloušťku. Toho se docílí pomocí kolmého vektoru ke směru stěny, což je vektor mezi počátečním a koncovým bodem stěny. Tento kolmý vektor je vždy podle orientace stěny vlevo (viz Obrázek 5.8). Dále mohou stěny obsahovat buď dveře, nebo okna.

Obrázek 5.8: Směr rozšíření stěny.

Dalším krokem je vytvoření podlah pater a podezdívku. Pro vytvoření podezdívky jsou získány nejvyšší a nejnižší výška terénu generované parcely. Dle těchto výšek je určena výška podezdívky jako rozdíl těchto výšek. K nejvyšší výšce je navíc přičtena konstanta daná pomocí použité konfigurace. Podlahy budov i podezdívka jsou generovány z rozměrů a pozice parcely, na které leží.

Obrázek 5.9: Budova vytvořená generátorem.

6 Závěr

Podařilo se navrhnout a implementovat aplikaci generující a zobrazující terén a město. Město se generuje na základě vstupních parametrů vstupní semínko a vzoru silniční sítě. Generátor je plně deterministický, takže při stejně zadaných vstupních parametrech generuje stejné výsledky. Výsledné modely se zobrazí ve 3D scéně, kterou je možné procházet. Terén je vygenerován mírně kopcovitý. Podle zadaného vzoru je vygenerován silniční systém, který se snaží napodobovat vzory silničních sítí, jenž lze pozorovat při pohledu do map reálných měst. Podle silničních sítí jsou vygenerovány bloky a ty dále rozděleny na parcely. Na těchto parcelách jsou vytvořeny budovy. Modely budov jsou vygenerovány včetně interiérů. Mezi místnostmi jsou dveře umístěny takovým způsobem, aby bylo možné se z každé místnosti dostat do vstupní haly.

V závislosti na zadaném vzoru silničních sítí se čas generování liší. Při použití mřížkového vzoru jsou bloky a parcely obdélníkové a díky čemuž se vytváří velké množství budov. Čas potřebný pro vygenerování celé scény je přibližně 20 vteřin. Průměrně je v rámci jedné scény vytvořeno 1700 budov, z nichž každá se skládá průměrně z 1396 vrcholů a 3071 trojúhelníků.

V budoucnu by se generované modely budov, silnic, a terénu mohli opatřit texturami, například také generovanými pomocí procedurálních metod. Zajímavé by jistě také bylo generování modelů budov s větší rozmanitostí druhů, půdorysů a architektur. Těmito způsoby by se model více přiblížil k objektivní realitě světa, což by mohlo být námětem pro další práce v oblasti grafického modelování.

Literatura

1. **Hugo, Elias.** Perlin noise. [Online] [Citace: 16. Květen 2016.] http://freespace.virgin.net/hugo.elias/models/m_perlin.htm.

2. **Bourke, Paul.** Interpolation Methods. [Online] Prosinec 1999. [Citace: 7. Červenec 2016.] http://paulbourke.net/miscellaneous/interpolation/.

3. **Finley, Darel Rex.** Ultra-Easy Polygon Area Algorithm With C Code Sample. [Online] 2006. [Citace: 10. Červenec 2016.] http://alienryderflex.com/polygon_area/.

4. **Greiner, Günther a Hormann, Kai.** Efficient clipping of arbitrary polygons. *ACM Transactions on Graphics.* Duben 1998, Sv. 17, 2, stránky 71-83.

5. **Lindenmayer, Aristid a Prusinkiewicz, Przemyslaw.** *The Algoritmic Beauty of Plants.* místo neznámé : Springer Science & Business Media, 2012.

6. **Chen, Guoning, Esch, Gregory a Wonka, Peter.** Interactive procedural street modeling. *ACM transactions on graphics (TOG).* 2008, Sv. 27, 3, str. 103.

7. **Bruls, Mark, Huizing, Kees a Van Wijk, Jarke J.** Squarified Treemaps. *Data Visualization 2000.* místo neznámé : Springer, 2000, stránky 33-42.

8. **Greuter, Stefan, Parker, Jeremy a Steward, Nigel.** Undiscovered Worlds--Towards a Framework for Real-Time Procedural World Generation. 2002.

9. **Lechner, Thomas, Watson, Ben a Wilensky, Uri.** Procedural City Modeling.

10. **Parish, Yoav IH a Müller, Pascal.** Procedural modeling of cities. *Proceedings of the 28th annual conference on Computer graphics and interactive techniques.* 2001, stránky 301-308.

11. **Vanegas, Carlos A., Kelly, Tom a Weber, Basil.** Procedural generation of parcels in urban modeling. *Computer graphics forum.* 2012, Sv. 31, 2pt3, stránky 681-690.

12. **Marson, Fernando a Musse, Soraia Raupp.** Automatic real-time generation of floor plans based on squarified treemap algorithm. *Internation Journal of Computer Games Technology 2010.* 2010, 7.

13. **Kelly, George a McCabe, Hugh.** A survey of procedural techniques for city generation. *ITB Journal.* Prosinec 2006, 14, stránky 87-130.

14. **Hormann, Kai a Agathos, Alexander.** The point in polygon problem for arbitrary polygons. *Computational Geometry.* 2001, Sv. 20, 3, stránky 131-144.

15. **Wright Jr, Richard S., Haemel, Nicholas a Sellers, Graham M.** *OpenGL SuperBible: comprehensive tutorial and reference.* Seventh edition. New York : Addison-Wesley, 2015. ISBN 978-0672337475.

16. **Müller, Pascal, Wonka, Peter a Heagler, Simon.** Procedural modeling of buildings. *ACM Transaction On Graphics (TOG).* 2006, Sv. 25, 3, stránky 614-623.

Příloha A

Obsah CD

Přiložené CD obsahuje zdrojové kódy a soubory aplikace, spustitelnou verzi aplikace ve formátu .exe, bakalářskou práci ve formátu .pdf i se zdrojovým souborem a soubory s návodem k aplikaci.

- ./ProceduralCityGeneration - složka obsahující veškeré zdrojové soubory
- ./Dokumentace - složka, která obsahuje tento dokument
- ./Video - složka obsahující prezentaci pomocí videa
- ./Aplikace - složka obsahující spustitelné soubory v prostředí Windows
- ./readme.txt - manuál pro ovládání aplikace

Příloha B

Manuál

Kompilace a Spuštění

Aplikaci lze sestavit za pomocí Visual Studia 2015. Soubor s projektem se nachází ve složce "ProceduralCityGeneration". Ke spuštění aplikace je třeba mít dostupný VS2015 Redistributable a grafická karta musí podporovat alespoň verzi OpenGL 3.3.
Aplikace po spuštění vyžaduje náhodné číslo a zvolení přednastavené konfigurace.

Ovládání

W - Pohyb dopředu
S - Pohyb dozadu
A - Pohyb bokem doleva
D - Pohyb bokem doprava
Shift - při držení zvýšení rychlosti
Escape - ukončení aplikace

Otáčení kamerou:
Držení levého tlačítka myši a pohyb myší